Quantitative Cyber-Security

Colorado State University
Yashwant K Malaiya

CS559

L19: Fuzzers, Pen Testing



CSU Cybersecurity Center
Computer Science Dept

Quantitative Security

Colorado State University
Yashwant K Malaiya

CS 559
Fuzzing



CSU Cybersecurity Center
Computer Science Dept

Fuzzing vs. Testing

User Testing for ordinary bugs

- Run program on many **normal** inputs, look for bad things to happen
- **Goal:** Prevent **normal users** from encountering errors

Fuzzing to find vulnerabilities

- Run program on many **abnormal** inputs, look for bad things to happen
- **Goal:** Prevent **attackers** from encountering exploitable errors

Probing input space:

- Vulnerabilities are very sparce
- Only 1-5% of defects are vulnerabilities
- Lot of testing to find vulnerabilities. Automation needed.

Ack: Stanford, Columbia

Types of Fuzzing

- **Mutation-based (Dumb) fuzzing**
  - Add anomalies to existing good inputs (e.g., test suite)
- **Generative (Smart) fuzzing**
  - Generate inputs from specification of format, protocol, etc
- **Evolutionary (Responsive) fuzzing**
  - Leverage program instrumentation, code analysis
  - Use response of program to build input set

Mutation-Based Fuzzing

Basic Idea

- Take known good input and add anomalies
- Anomalies may be completely random or follow some heuristics
  - Large integers or strings
  - Randomly flip bits

HTTP Fuzzing Example

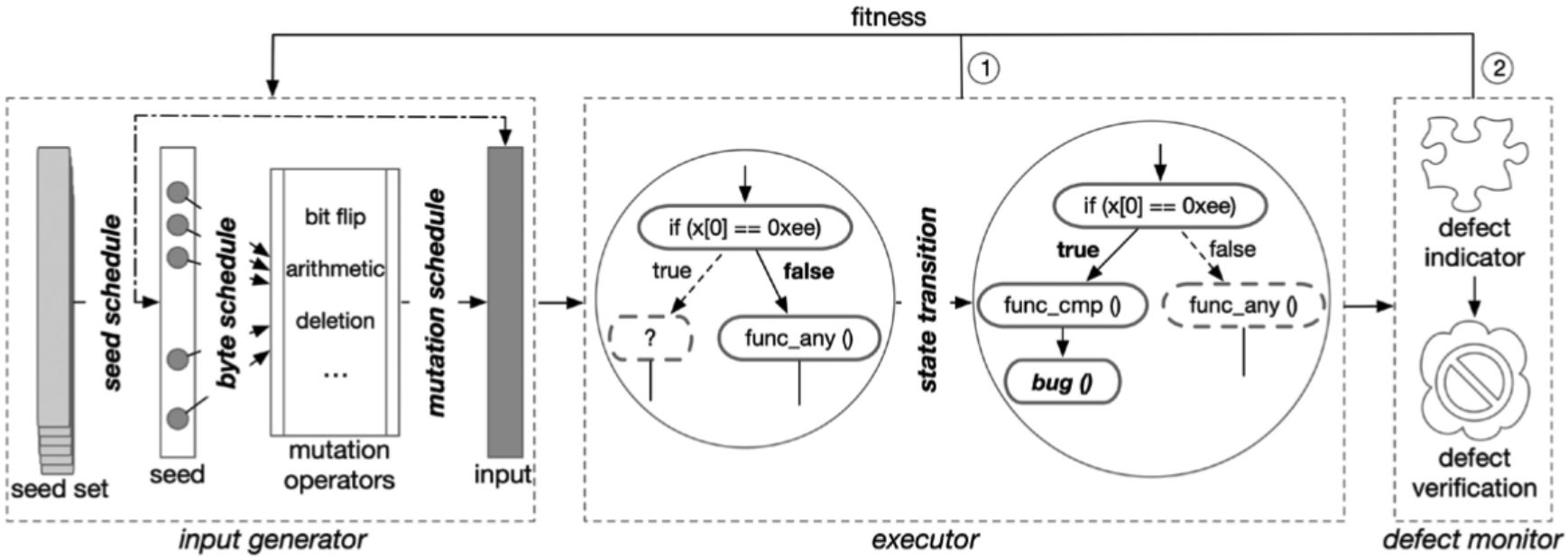
Standard HTTP GET Request

GET /index.html HTTP/1.1

Anomalous Requests

GEEEE…EET /index.html HTTP/1.1
GET ///////index.html HTTP/1.1
GET %n%n%n%n%n%n.html HTTP/1.1
GET /AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.html HTTP/1.1
GET /index.html HTTTTTTTTTTTTP/1.1
GET /index.html HTTP/1.1.1.1.1.1.1.1
df%w3rasd8#r78jskdflasdjf
4isg8swksdfskdflsdgmsf$gkjs

General workflow of fuzzing



General workflow of fuzzing. Essentially, fuzzing consists of three components: input generator, executor, and defect monitor.

An input is retained as a seed when the input achieves better fitness (e.g., new coverage). Later, fuzzing campaigns will select seeds in the seed set to perform mutations.

Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. ACM Comput. Surv. 54, 11s, Article 230 (January 2022), 36 pages. https://doi.org/10.1145/3512345

Bugs that are detected by fuzzing

- **Memory-violation Bugs** can be classified into two categories, i.e., spatial violations and temporal violations. A spatial violation gets access to memory that is out of bounds, while a temporal violation accesses an invalid referent.
- **Concurrency Bugs** which occurs when concurrent programs run without proper synchronization or order.
- **Algorithm complexity (AC) vulnerabilities** occur when the worst-case complexity of an algorithm significantly reduces the performance, which could result in Denial-of-Service (DoS) attacks.
- **A Spectre-type bug** is a microarchitectural attack that exploits mispredicted branch speculations to control memory accesses.
- **Side-channel bugs** leak secret information via observing non-functional behaviors of a system (e.g., execution time)
- **Integer overflow/underflow** bugs occur when the value of an arithmetic expression is out of the range that is determined by a machine type. Also integer conversion bugs.

Fuzzing PDF Reader

- Download 100s of random PDF files
- Mutate content in the PDF file:
  - flip bits
  - increase size of integers or strings
  - remove data
- Limited by the functionality that the existing files happened to use — unlikely to hit less commonly tested code paths





Mutation-Based Fuzzing

Basic Idea

- Take known good input and add anomalies
- Anomalies may be completely random or follow some heuristics

Advantages

- Little or no knowledge of the structure of the inputs is assumed
- Requires little to no set up time

Disadvantages

- Dependent on the inputs being modified
- May fail for protocols with checksums, challenge-response, etc.

Generation Based Fuzzing

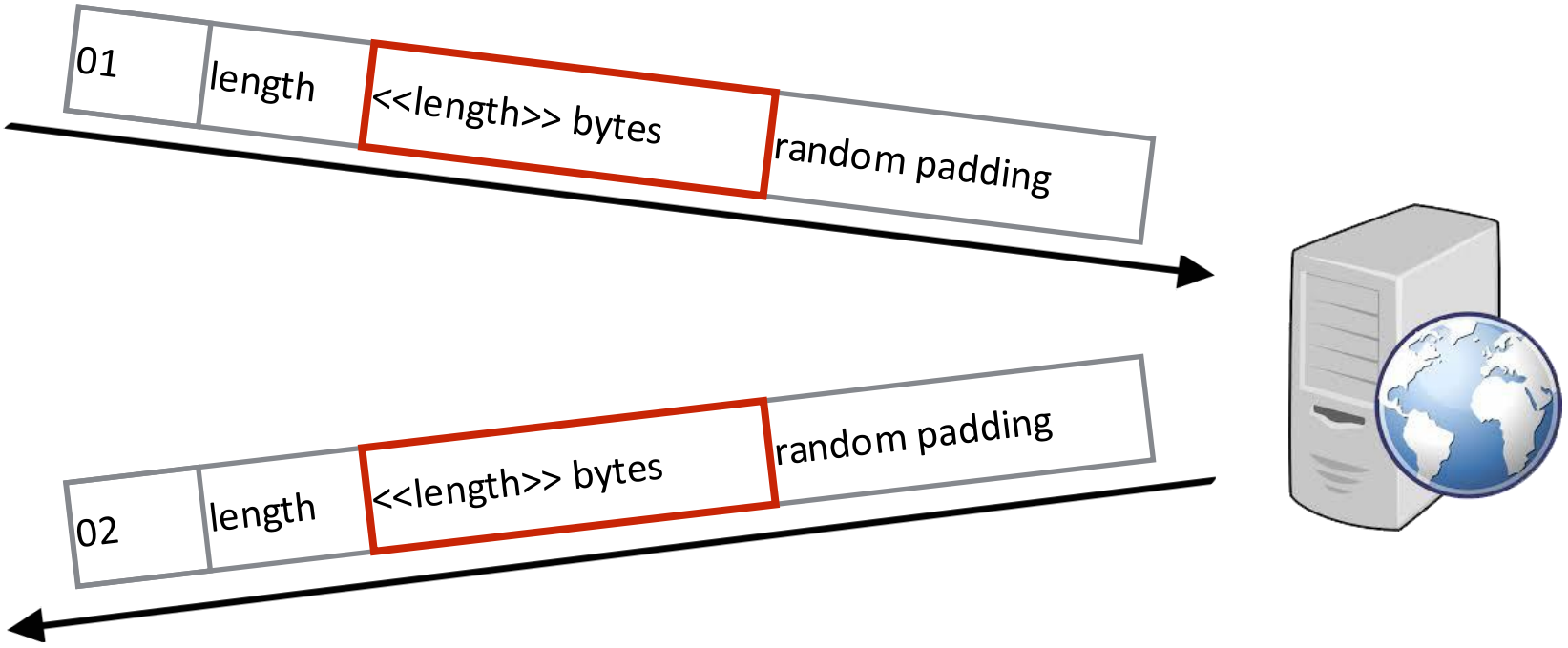
Basic Idea

- Test cases are generated from protocol description: RFC, spec, etc.
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing

```
<!--  A.  Local file header -->
  <Block name="LocalFileHeader">
    <String name="lfh_Signature" valueType="hex" value="504b0304" token="true" mut
    <Number name="lfh_Ver" size="16" endian="little" signed="false"/>
    ...
    [truncated for space]
    ...
    <Number name="lfh_CompSize" size="32" endian="little" signed="false">
      <Relation type="size" of="lfh_CompData"/>
    </Number>
    <Number name="lfh_DecompSize" size="32" endian="little" signed="false"/>
    <Number name="lfh_FileNameLen" size="16" endian="little" signed="false">
      <Relation type="size" of="lfh_FileName"/>
    </Number>
    <Number name="lfh_ExtraFldLen" size="16" endian="little" signed="false">
      <Relation type="size" of="lfh_FldName"/>
    </Number>
    <String name="lfh_FileName"/>
    <String name="lfh_FldName"/>
    <!--  B.  File data -->
    <Blob name="lfh_CompData"/>
  </Block>
```

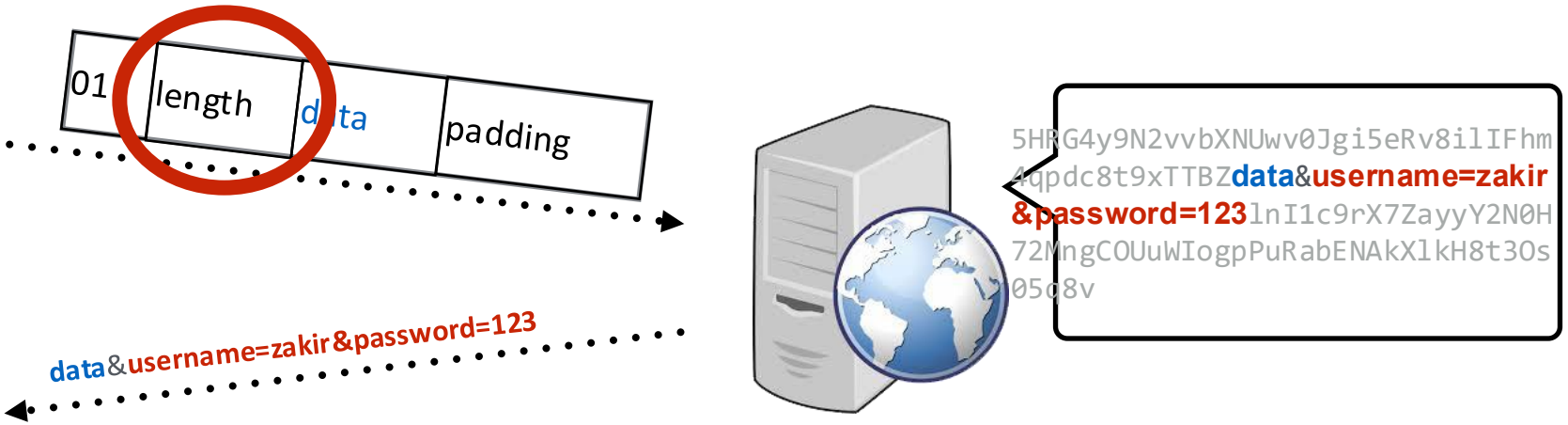
Generation Example: TLS Heartbeat

Heartbeat Extension for the Transport Layer Security:
to test and keep alive secure communication links
without the need to renegotiate the connection each time



Colorado State University

Generation Example: TLS Heartbeat



Heartbleed Vulnerability: server trusts user provided length field and echoes back memory contents following request data

Mutation-based vs. Generation-based

- Mutation-based fuzzer
  - Pros: Easy to set up and automate, little to no knowledge of input format required
  - Cons: Limited by initial corpus, may fall for protocols with checksums and other hard checks
- Generation-based fuzzers
  - Pros: Completeness, can deal with complex dependncies (e.g, checksum)
  - Cons: writing generators is hard, performance depends on the quality of the spec

How much fuzzing is enough?

- Mutation-based-fuzzers may generate an infinite number of test cases. When has the fuzzer run long enough?
- Generation-based fuzzers may generate a finite number of test cases. What happens when they're all run and no bugs are found?
- Sometimes every anomalous test case triggers the same (boring) bug?

Charlie Miller's 5 Lines

In 2010, Charlie Miller fuzzed

- Adobe Acrobat,
- Apple Preview,
- Powerpoint, and
- Open Office

by downloading PDF and PPT files and five lines of simple fuzzing:

```
numwrites = random.randrange(math.ceil((float(len(buf)) /
FuzzFactor))) + 1
for j in range(numwrites):
  rbyte = random.randrange(256)
  rn = random.randrange(len(buf))
  buf[rn] = "%c"%(rbyte)
```

Charlie Miller's 5 Lines

Collect a large number of pdf files

- Aim to exercise all features of pdf readers
- Found 80,000 PDFs on Internet

Reduce to smaller set with apparently equivalent code coverage

- Used Adobe Reader + Valgrind in Linux to measure code coverage
- Reduced to 1,515 files of 'equivalent' code coverage (Test compaction)
- Same effect as fuzzing all 80k in 2% of the time

Randomly changed selected bytes to random values in files

- Produce ~3 million test cases from 1,500 files

Use standard common tools to determine if crash represents a vulnerability

- Acrobat: 100 unique crashes, 4 actual vulnerability 4%
- Preview: 250 unique crashes, 60 exploits (tools may over-estimate) 24%

Code Coverage

What if we tried to build tests that try to reach code in the program?

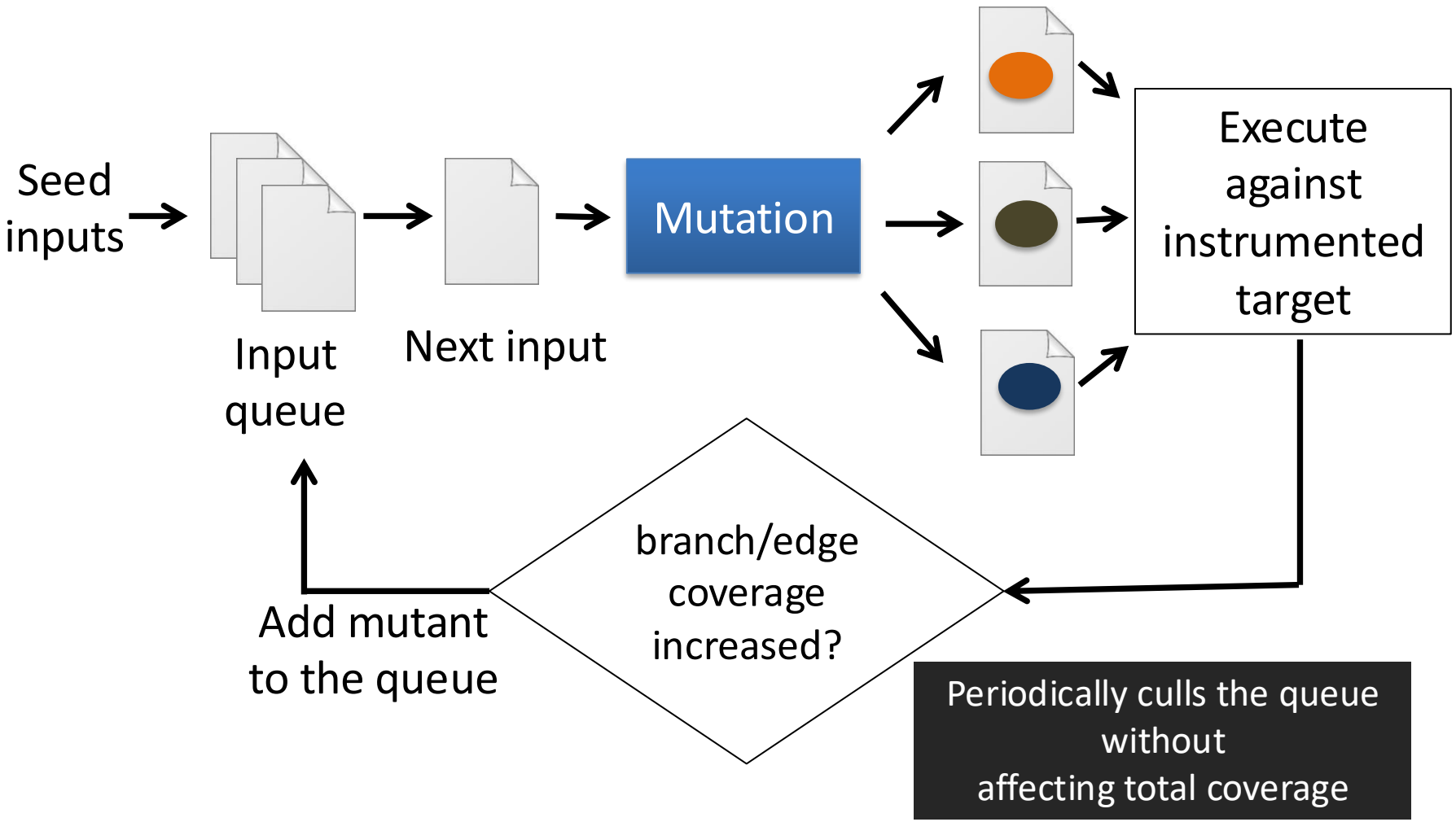
Code coverage is a metric which can be used to determine how much code has been executed.

- **Function coverage:** Has each function in the program been called?
- **Edge coverage:** Has every edge in the Control flow graph been executed?
- **Branch coverage:** Has each branch of each control structure been executed?
- **Predicate coverage:** Has each Boolean expression been evaluated to true and false?

Coverage-guided gray-box fuzzing

- Special type of mutation-based fuzzing
  - Run mutated inputs on instrumented program and measure code coverage
  - Search for mutants that result in coverage increase
  - Often use genetic algorithms, i.e., try random mutations on test corpus and only add mutants to the corpus if coverage increases
  - Examples: AFL, libfuzzer

American Fuzzy Lop (AFL)



Evolutionary Fuzzing

Basic Idea:

Generate inputs based on the structure and response of the program

- **Autodafe:** Prioritizes based on inputs that reach dangerous API functions
- **EFS (**Evolving **Fuzzer** System**):** Generates test cases based on code coverage metrics

Typically instrument program with additional instructions to track what code has been reached — or, if no source is available, track with Valgrind.

Tools

Two influential tools

cross_fuzz — specifically targeted at browser and generating complex DOM (document object model) sequences

American Fuzzy Lop (AFL) — most everything else

AFL Algorithm

American fuzzy lop (AFL) 2013 initial /2019 stable Michał Zalewski, Google/Snap

- Load user-supplied initial test cases into the queue,
- Take next input file from the queue,
- Attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program,
- Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies,
- If any of the generated mutations resulted in a new state transition recorded by the instrumentation,
  - add mutated output as a new entry in the queue.
- Go to 2.

Fuzzing challenges

- How to seed a fuzzer?
  - Seed inputs must cover different branches
  - Remove duplicate seeds covering the same branches
  - Small seeds are better.

- Some branches might be very hard to get past as the # of inputs staisfying the conditions are very small
  - Manually/automatically transform/remove those branches

Fuzzing rules of thumb

- Input-format knowledge is very helpful
- Generational tends to beat random, better specs make better fuzzers
- Each implementation will vary, different fuzzers find different bugs
  - More fuzzing with is better
- The longer you run, the more bugs you may find
  - But it reaches a plateau and saturates after a while
- Best results come from guiding the process
- Notice where you are getting stuck, use profiling (gcov, lcov)!

Evaluating Fuzzers

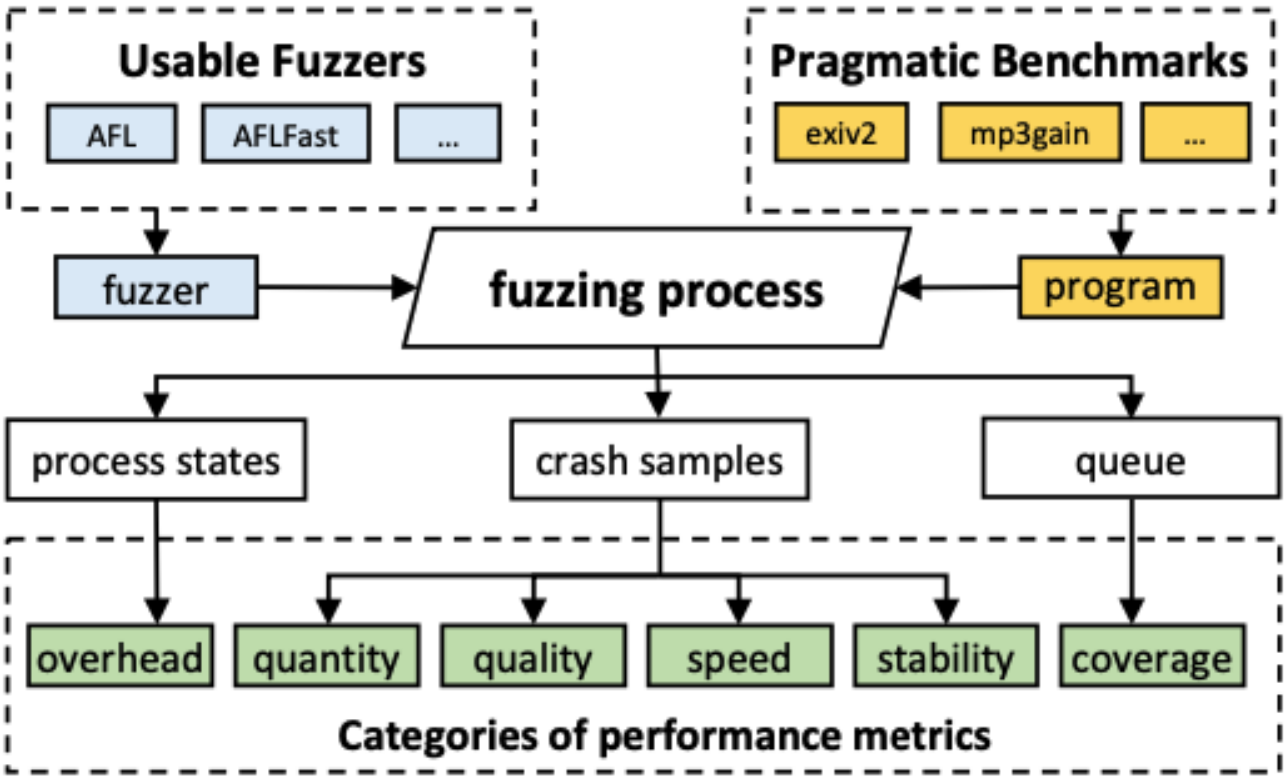


Figure 1: Overview of UNIFUZZ.

Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. Yuwei Li, et al, USENIX Security Symposium 2021: 2777-2794.

Evaluating Fuzzers

- 35 fuzzers, 20 benchmark programs
- Performance metrics: Number of unique Bugs, severity, detectability, Speed of Finding Bugs, coverage, computational overhead (memory, disk etc)
- Best ones: Honggfuzz (Google, 2017), MOPT (Lyu et al, 2019), QSYM (Yun et al, 2018)

The average number of unique *EXPLOITABLE* bugs

Table 5: The average number of unique *EXPLOITABLE* bugs.

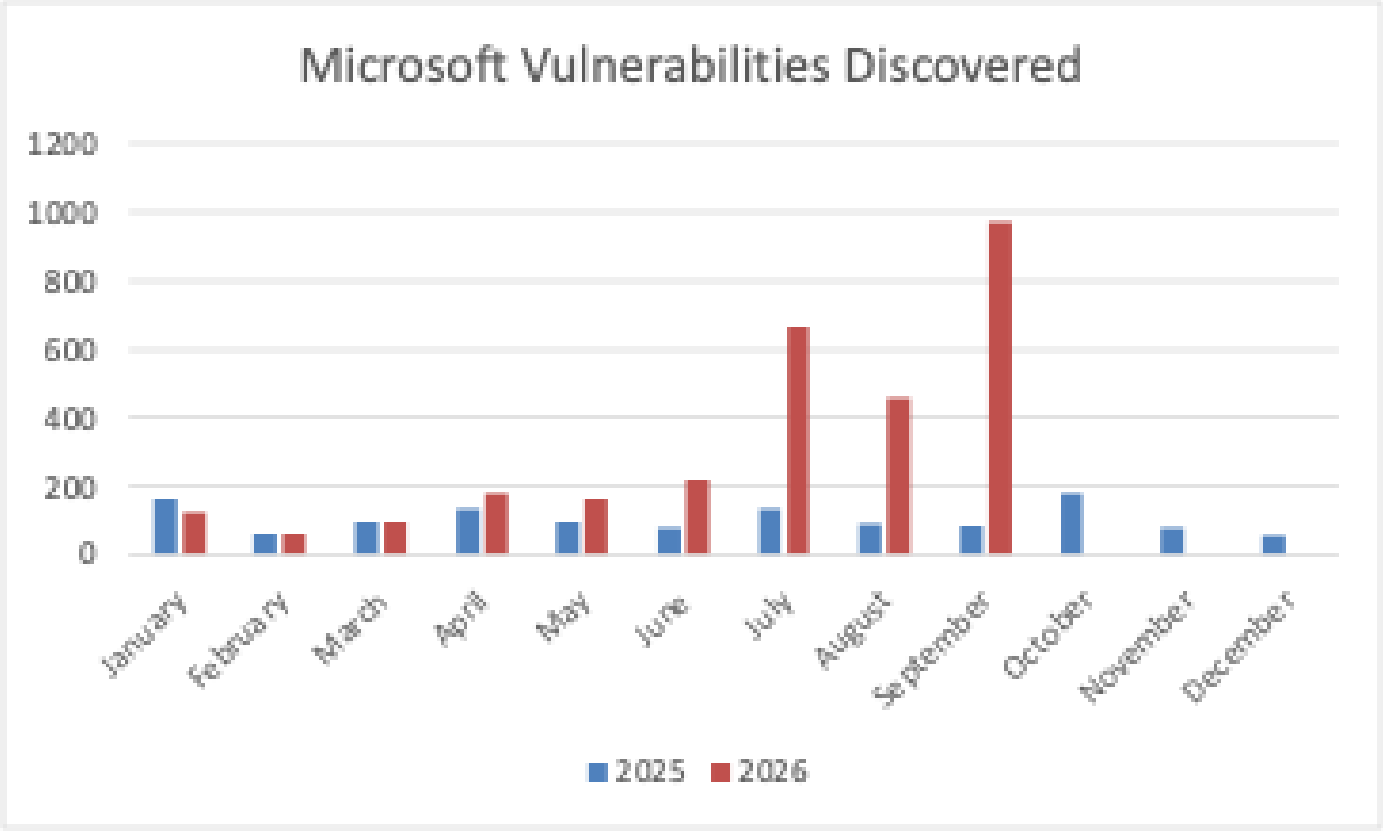
	AFL	AFLFast	Angora	Honggfuzz	MOPT	QSYM	T-Fuzz	VUzzer64
exiv2	1.3	0.5	**6.7**	0.1	4.9	0.3	0	0
gdk	0.0	0.3	1.2	0	**7.9**	2.3	4.7	0.5
imginfo	0	0	0	0	0	**0.03**	0	0
jhead	0	0	**0.2**	0	0	**0.2**	**0.2**	0
tiffsplit	0.7	**0.8**	0.2	0	**0.8**	0	0.7	0
lame	1.0	1.0	5.8	3.4	**9.2**	3.1	1.0	0
mp3gain	0.1	0.1	0	**2.0**	0.8	0.8	0	0
wav2swf	3.0	3.1	3.0	0.1	**10.0**	3.0	0	0.3
ffmpeg	0	0	0	**0.1**	0	0	n.a.	0
flvmeta	0.2	0.3	0.1	**0.6**	0.5	0.1	0.1	0
mp42aac	0	0	0.0	0	0	**0.5**	0	0
cflow	0	0	0	**0.8**	0.2	0	0	0
infotocap	0	0	0	0	**0.2**	0	0	0
jq	0	0	0	0	0	0	0	0
mujs	0	0	0	**0.1**	0	0	0	0
pdftotext	0.3	0.7	1.0	0.7	**7.2**	5.0	0	0
sqlite3	0	0	0	0	**3.1**	0	0	n.a.
nm	0	0	**4.8**	0	0	0	0	0
objdump	0.3	0.4	1.2	0.5	**3.2**	0	0.2	0
tcpdump	0	0	0	0	**0.3**	0	0	0

New Cyber security Crisis

- **The "PHP2Shell" Discovery**
- Researchers at OpenAI, Wiz, and Calif used frontier AI models (GPT-5.6 Sol and Gemini 4 Argon) to discover a catastrophic series of vulnerabilities hidden in PHP code for over a decade.
- CVE-2026-109618 and CVE-2026-109619 both with CVSS of 9.2. Allow remote code execution.
- The flaw affects roughly 70% of the internet and has forced many websites to go offline today to patch it.

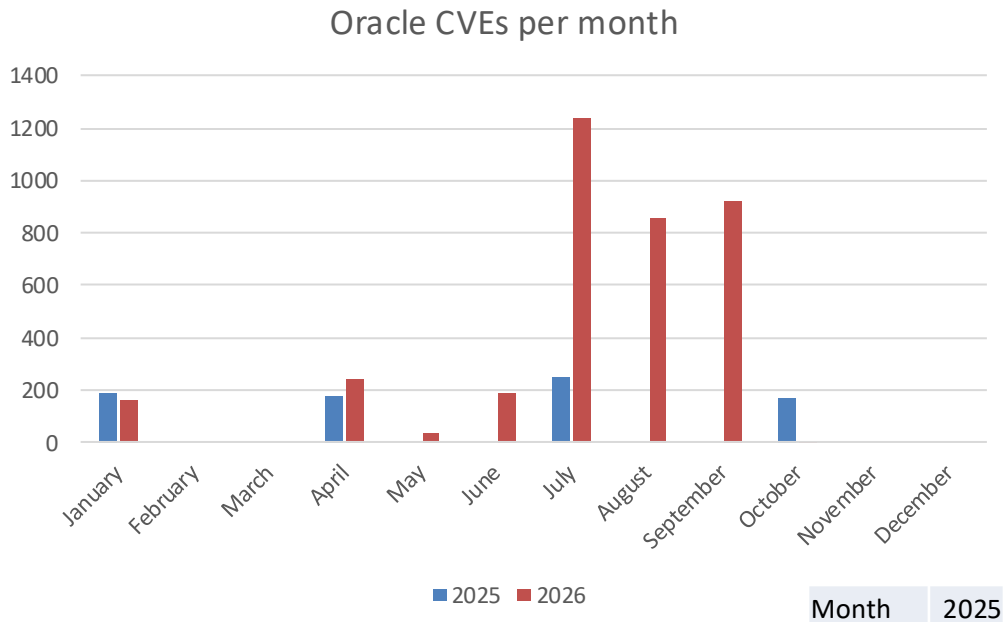
AI for Vulnerability Discovery

- Use of AI has dramatically speeded up vulnerability discovery.



Colorado State University

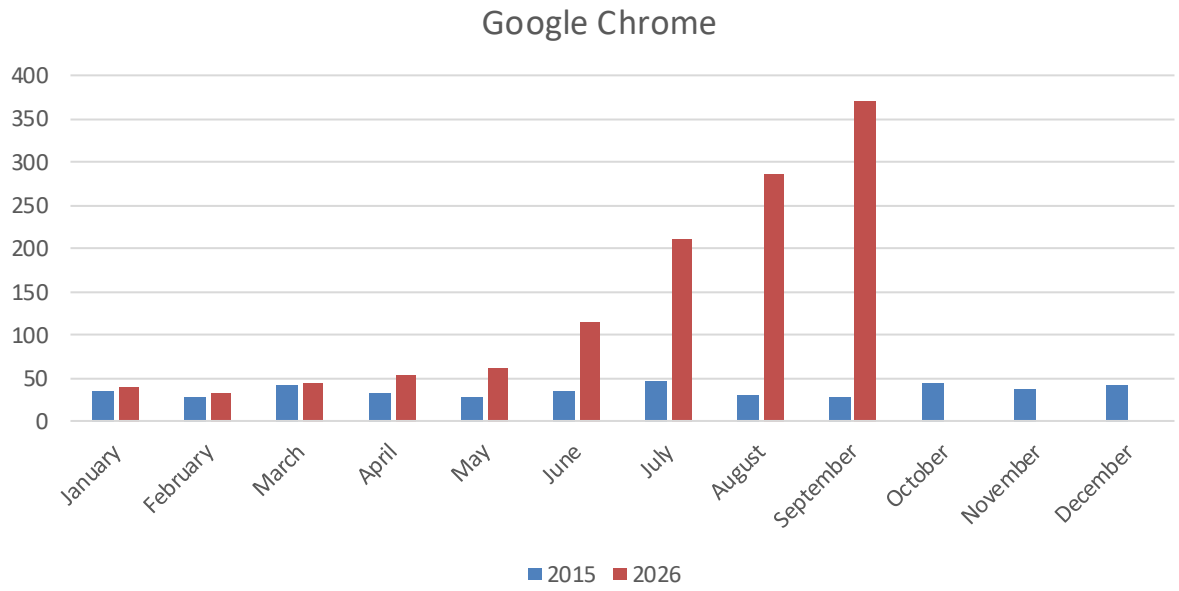
Oracle Vulnerabilities



Month	2025	2026	2025	2026
January	186	158	Quarterly CPU Rollup	Quarterly CPU Rollup
February	0	0	No Scheduled Release	No Scheduled Release
March	0	0	No Scheduled Release	No Scheduled Release
April	171	241	Quarterly CPU Rollup	Quarterly CPU Rollup
May	0	35	No Scheduled Release	Inaugural Monthly CSPU
June	0	192	No Scheduled Release	Monthly CSPU Rollup
July	248	1235	Quarterly CPU Rollup	Quarterly CPU Rollup (AI Surge)
August	0	860	No Scheduled Release	Monthly CSPU Rollup
Sept	0	920	No Scheduled Release	Monthly CSPU Rollup
October	170	9	Quarterly CPU Rollup	
Nov	0	0	No Scheduled Release	
Dec	0	0	No Scheduled Release	

Colorado State University

Google Chrome Vulnerabilities



2026	January	39	Standard early-year channel baseline.
	February	32	Standard channel updates; zero-day CVE-2026-2441 patched.
	March	45	Emergency releases for two active zero-days (CVE-2026-3909/3910).
	April	52	First major volume uptick following public frontier model integrations.
	May	61	Sustained elevated volume over historic baselines.
	June	114	AI-guided fuzzing and continuous automated tracking loops mature.
	July	210	Massive structural surge driven by internally deployed Gemini/Agent frameworks.
	August	285	High-volume rollout from autonomous agentic code sweeps.
	Sept	370	Record-breaking stable release cycles (Chrome 151) addressing a wave of automated findings.

AI Fuzzing vs. Traditional Fuzzing

- Traditional fuzzers (like AFL++ or libFuzzer) operate essentially by "blindly" flipping bits or randomly mutating input files.
- AI fuzzing is fundamentally better due to the following structural differences.

AI Fuzzing vs. Traditional Fuzzing

Capability	Traditional Fuzzing	AI-Powered Fuzzing
Harness Engineering	Manual: Humans must write the complex code ("fuzz targets") to feed input into an API.	Automated: AI automatically generates syntactically correct, structure-aware fuzz targets at massive scale.
Input Structure (The Seed Pool)	Stochastic / Random: Mutates inputs via random bit-flipping. If a program requires a strict file format the fuzzer gets blocked.	Semantic & Intelligent: AI understands code logic and generates complex, syntax-compliant inputs that bypass early input-validation checks.
Code Coverage	Shallow Depth: Fails to reach hidden or deeply nested logic blocks.	Deep Penetration: Achieves drastically higher code coverage. Google's AI-fuzzing frameworks successfully forced testing tools into obscure code paths across 272 projects that human-written setups never reached.
Target Application	Limited to Binaries: Primarily constrained to compiled code, memory limits, and command-line inputs.	Expands to Natural Language: Extends to testing modern AI models. AI fuzzers can intelligently mutate text prompts to systematically find "jailbreaks" and safety guardrail bypasses.

How AI Accelerates Overall Vulnerability Discovery

- Traditional vulnerability discovery relies on human security researchers manually inspecting source code, reverse-engineering binaries, or setting up static application security testing (SAST) tools that flag thousands of false positives.
- AI speeds this process up.
  - **Autonomous Setup & Orchestration**: AI agents can analyze an entire software library, instantly map out its public APIs, configure build environments, and write the custom testing harnesses required saving hundreds of hours.
  - **Contextual Triaging**: When a security flaw found, AI models can instantly analyze the application's active state, deduce exactly what type of bug it is (e.g., a buffer overflow vs. a use-after-free error), and trace the data flow to prove if it is exploitable.
  - **Closing the Loop**: Advanced platforms create autonomous pipelines. Google's CodeMender, for instance, links AI discovery tools to an LLM that autonomously drafts, tests, and upstream merges a security fix directly back into the codebase

Tool Pipeline

These Google tools function as a unified pipeline:

- **OSS-Fuzz** continuously tests the codebase and generates a crash report.
- **Big Sleep** analyzes the crash context to trace exactly how a malicious actor could exploit it.
- **CodeMender** automatically uses that data, creates a patch, validates that it doesn't break the application, and pushes a pull request directly to the developers.

AI-driven remediation

Patching Phase	Traditional / Without AI	Autonomous / With AI
Typical Enterprise Cycle	30 to 90 days (Governance SLAs typically give 30 days for critical, 60 for high)	6 to 13 days (Autonomous endpoint patching tools)
Average Real-World MTTP	43 days (up from 32 days in 2024 due to increased complexity)	Under 3 days (When integrating AI scanners with virtual patching/WAAP solutions)
Testing & Verification	8 to 14 days of traditional siloed testing	Automated validation and immediate deployment orchestration

Defensive asymmetry

AI has broken traditional, patch-centric vulnerability management.

- **Centralized Bottlenecks: The National Vulnerability Database (NVD) Crisis** NVD, has been overwhelmed by record-breaking CVE growth driven heavily by AI-assisted submissions. Of the more than 53,000 CVEs published by late 2026, nearly 79% sit in an "Awaiting Analysis" backlog. Roughly 80% of new CVEs are now labeled "Not Scheduled.
- **The Open-Source & Maintainer Chokepoint** Anthropic's Project Glasswing utilized AI to surface 26,153 vulnerabilities. Out of more than 2,000 officially disclosed to maintainers, fewer than 1% have actually been patched.
- **Enterprise Triage Overwhelm** :
  - **Signal vs. Noise:** Poorly implemented AI code scanners routinely drop massive lists of unvalidated flaws into development backlogs, sometimes yielding true-positive rates below 7%.
  - **Fixing vs. Breaking:** more than 53% of AI-generated patches either failed to fix the core vulnerability, broke application logic entirely, or introduced brand-new security flaws.

Quantitative Security

Colorado State University

Yashwant K Malaiya

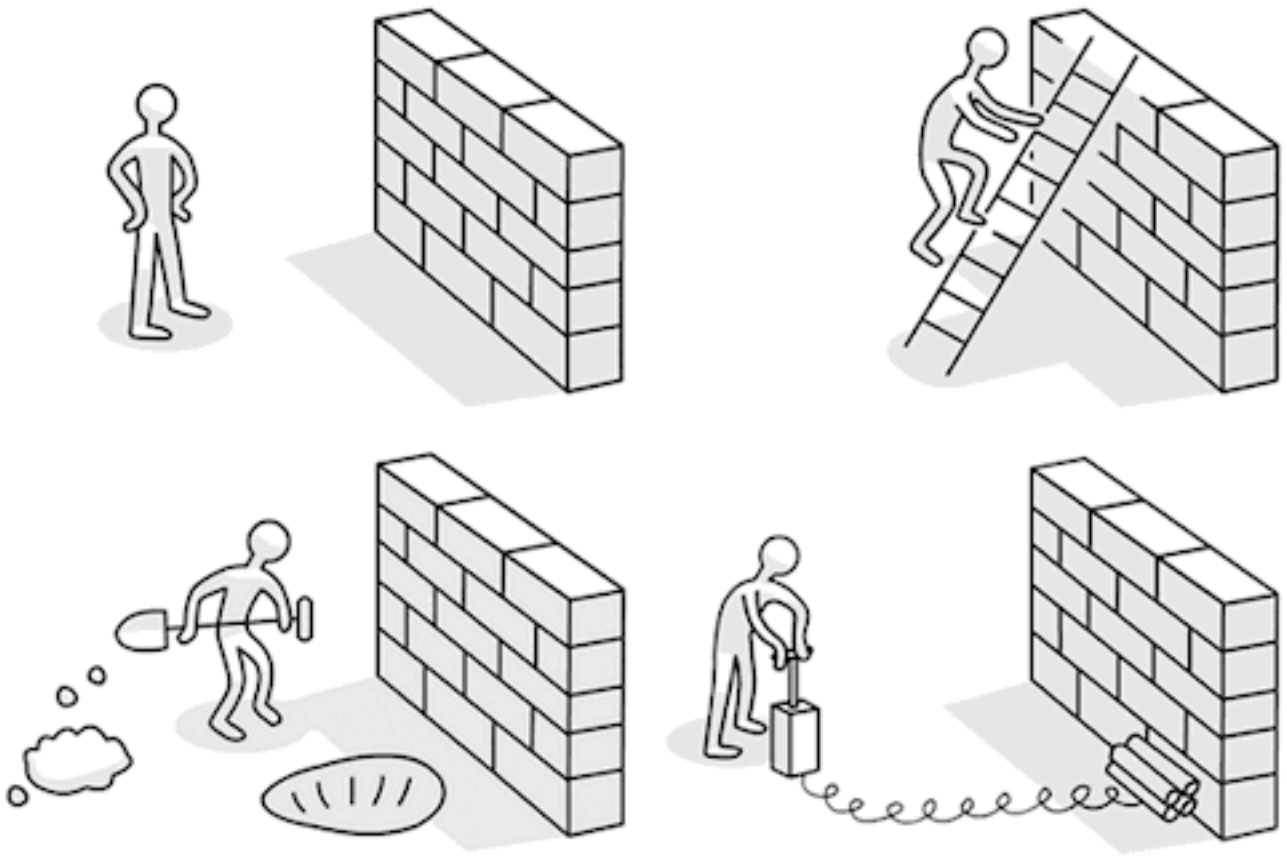
CS 559

Penetration Testing



CSU Cybersecurity Center
Computer Science Dept

Penetration Testing



https://www.cobalt.io/blog/how-to-choose-a-pentest-service

Colorado State University

Penetration Testing

How to get into a system and steal stuff

- Not legal in general
- Completely legal if you are hired and authorized to do Pen Testing. Very valuable service.
- Involves all kinds of bad/fun stuff.
- A **penetration test**, or a **pen test**, is a simulated cyber attack against a computer system (or enterprise) to check for exploitable vulnerabilities.

Teaching How to Pick a Pocket or Two



- Fagin in Charles Dicken‘s Oliver Twist (1838)
- https://www.youtube.com/watch?v=2YVAmZBGdXw Fifth Av Theater

- A remote area in Jharkhand's Maoist-hit **Jamtara** district, a man runs a coaching centre where he teaches his pupils trader tricks of online fraud

Good Guys gone Bad

- Nov 3, 2025: Two former employees of cybersecurity firms that sold services helping companies combat hackers have been indicted and accused of participating in a conspiracy, outside of their day jobs, to hack multiple US firms and extort them for millions of dollars.
- M worked for DigitalMint, an Illinois-based firm that helps victims recover from ransomware attacks and in some cases pays ransoms, according to its website. G worked for Sygnia Cybersecurity Services, a multinational firm whose offerings include simulating ransomware attacks for clients.

Quantitative Security

Colorado State University

Yashwant K Malaiya

CS 559

Penetration Testing



CSU Cybersecurity Center
Computer Science Dept

What is Penetration Testing?

- Definition: A penetration test is a method of evaluating the security of a computer system and/or network by simulating an attack from a malicious source
  - malicious source: also known as a Black Hat Hacker
- A Pen Tester vs a Hacker
  - Prior approval
    - Pen Tester's have prior approval from Senior Management.
    - Hackers need no approval.
  - Technical Skills and Tools
    - A Pen Tester's uses his technical skills & tools to identify weaknesses that needs fixing.
    - A hacker wants to exploit weaknesses for profit or satisfaction
  - Social Engineering Skills
    - Pen Tester's social engineering penetration attempts are there to raise awareness
    - Hackers social engineering attacks are to steal/damage data

Ack: NASA-Ames, 2010